SOLID

enjoy Development

組込み初 実行時アドレスバグ 自動検出機能

～不具合は現行犯で逮捕せよ！～

【第一章】どんなバグが検出できるのか？

SOLID 開発プラットフォームでは、不具合を作り込まない仕組み、不具合をスマートに検出するための様々な仕組みを持っているのが特徴です。これらは、デスクトップソフトウェア開発環境として実績のあるものを、SOLID が組込みソフトウェア開発環境として初めて利用できるようにしたものです。

不具合を作り込まないようにするには、SOLID-IDE が持っているインテリセンス機能（コーディング支援）や、コンパイラの自動文法チェック機能（コンパイラのバックグラウンド実行により文法エラーを指摘）が有効です。また SOLID ではプログラムの流れを論理的に解析して不具合を検出できる静的解析機能も標準で備えており、簡単な操作だけで利用可能です。

机上検証の後、実際にプログラムを動作させてはじめて現れる不具合の中で、ソフトウェア開発者が意図せず巻き込まれてしまう最も多いものがアドレスバグと称される「間違ったアドレスのメモリへのリード／ライト操作でデータを壊してしまう」ケースです。
SOLID にはこのアドレスバグを自動的に見つけるための実行時バグ検出機能を持っており、うっかりミスなどを効率よく検出しソフトウェアの品質向上に大変有効です。

本書では、SOLID の実行時バグ検出機能であるアドレスバグの自動検出を詳しく解説していきます。

第一章は、「どんなバグが検出できるのか？」についてご紹介します。第二章は、アドレスバグ自動検出の仕組みについて、第三章ではアドレスバグ検出機能の有効な使い方について、ご紹介します。

ソフトウェア開発者が困っていること

ソフトウェアの大規模化にともない、バグ探しが大掛かりになって大変

ソフトウェアの規模が大きくなると、複数メンバーによる分担開発はもちろん、協力会社に委託をして共同開発するというスタイルが当たり前になっています。そんな大規模ソフトウェアの開発で最も困ってしまうのは、「単体テストでは問題無く動作していたが、結合テストで不具合が起きる」というシーンではないでしょうか。

開発対象のソフトウェアの本来機能の仕様やアルゴリズムに関する不具合であれば、仕様に従って論理的に追いかける事が有効ですが、意図せず不当にメモリが書き換わっていたようなアドレスバグの場合は、目の前で起きている不具合現象は分かるものの、その原因となるバグがどこに潜んでいるのかを探すのが大掛かりになってしまいます。

- 開発関係者が皆で集まり、合同デバッグする事は現実的に可能ですか？
- 全てのソースコードを開発者で共有できますか？
- ソフトウェアの改訂によって、不具合の発生パターンが変わってしまい、バグの追い込みが振り出しに戻る事はありませんか？
- 結合テストを実施するためのハードウェア環境を、バグ解析のために長期間使用できますか？
- 製品リリースまでに十分な期間はありますか？

不具合が出ている実機にデバッガを接続し、不具合現象が起きているソースプログラムを見ながら怪しい箇所に片っ端からブレークポイントを張り、break > step in > step out > step over を繰り返しているうちに、不具合の出方が変わってしまう、という辛い「モグラ叩き」を延々と続けた経験をされた方もいらっしゃることとでしょう。

また、アドレスバグとなった動作が実行されたあと、その影響が不具合となって表面化するまで大量のコードが走ってしまうので、実行履歴（トレース）を延々と追いかけなければいけない事もめずらしくはありません。しかもトレース情報をバッファするメモリ容量が有限なので、問題の原因となった実行箇所は既にその後の履歴に上書きされてしまい、結局肝心な問題箇所が判らない事もあります。

意図せずにうっかり作り込んでしまったアドレスバグは、作り込んでしまった側が「見つけてもらうのを待つ」しかないのでしょうか？

SOLID が自動検出できるアドレスバグの例

SOLID のアドレスバグの自動検出機能は、従来のデバッグ手法のような、バグのありそうなところに当たりをつけてブレークポイントを張って、トレース履歴を解析したり、コツコツと step 実行をしながらバグを追い込んでいくという手法ではありません。SOLID 環境下でビルド～デバッグを行う一連の開発作業の中で、SOLID がバグ検出のためのランタイムを生成し、それを実行するだけで自動検出機能を簡単に利用できるのが大きな特徴です。

また SOLID がアドレスバグを検出するのは、不当アクセスが発生した瞬間です。たとえそのバグが他のプログラムに何の影響を与えていなかったとしてもバグが検出できるため、単体テストの段階でも効果的に利用できます。

以下に SOLID が自動検出できるアドレスバグの例をご紹介します。

スタックオーバーフロー

SOLID では、アドレスバグの代表的なものである、スタックオーバーフローの自動検出が可能です。スタックオーバーフローは異常な多重割り込みや、内部変数等のスタック情報の増加が原因で、設計時の想定以上のスタックを消費してしまい、スタックメモリの上限を突き抜けて他の領域にスタックを積んで（データを書き込んで）しまうという、組込みではよく目にするバグです。

SOLID では、ビルド時にツールチェーンで指定されたスタックメモリのアドレス範囲をツール側が把握しているので、デバッガを接続した状態でプログラムが動作中にスタックオーバーフローが発生すると、それを直ちに検出します。

スタックオーバーフローが発生した場合は、図 1-1①のように、 ”Data Abort Exception caused by stack overflow” とデバッグ例外ウインドウに表示し、プログラムの続行を中断（ブレーク）します。またその時に RTOS ビューワー（図 1-1②）ではタスクに割り当てられたスタックの消費量をビジュアルに表示するので、どのタスクがスタックオーバーフローの要因になったのかを容易に知ることができます。

スタックオーバーフローはソースプログラムの静的解析により検出する事は困難なので、このような自動検出機能は大変有効です。しかも SOLID では特別な設定や、実行時間のオーバーヘッド無しでスタックオーバーフローが検出できるというところも大きな利点といってよいでしょう。

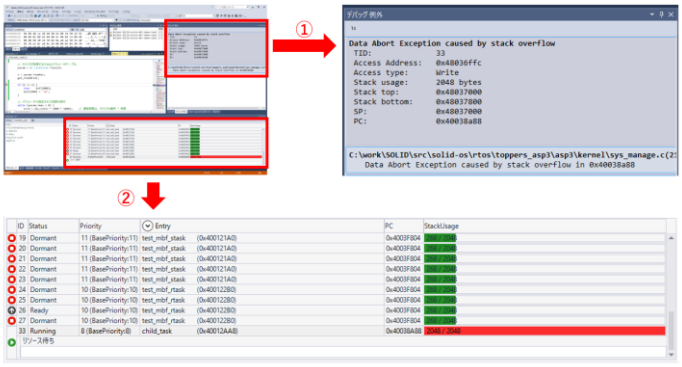


図 1-1 スタックオーバーフロー検出時のデバッガ画面表示

不当アクセス

ソプログラムのコード領域へ不当書き込みが起きると、本来動作するはずのプログラムコードが変わってしまうため「プログラムが暴走」してしまいます。不具合現象が発生したとき、それが暴走によるものかどうかを判断することは容易ではありませんし、暴走しているためにソフトウェアの流れを追う事すら難しいというのが実態です。

また不当書き込みによりコード領域を破壊してしまうような不具合は、ソフトウェアの規模が大きくなるにつれて、「他人の開発しているモジュールのコードを破壊する」という、自分では気づきにくいやっかいなバグです。

そこで SOLID では、プロセッサのアクセス例外検出機能を利用して不当アクセスを検出できるようにしました。以下がその動作概要です。

まず作成されたプログラムは、ビルド時に設定したプログラムのセクション情報に従い、ロードモジュールを生成するときに、
.text、.rodata、.data、.bss
という属性の異なるセクションに分けられます（図 1-2）。

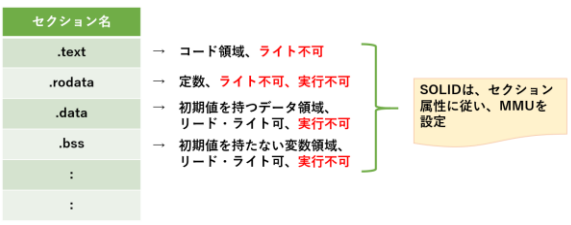


図 1-2. ビルド時に設定したセクション情報に従い、SOLID が MMU のメモリ属性を設定

各セクションは

.text は実行コードなので、「ライト不可」、「実行可能」
.rodata は read only データだから、「ライト不可」、「実行不可」

.data や .bss は、読み書き可能データだから、「ライト可」、「実行不可」

という属性を持っており、セクションの属性に従い SOLID が MMU の設定を行います。

MMU によりライト不可と設定されたコード領域へのライト動作が発生した場合には、メモリが書き換えられる前にプロセッサがアドレス例外を起こすので、SOLID デバッガと連動してバグ発生を検出しプログラムがブレークします。

実行不可領域でのプログラム実行も同様に検出します。

また、セクションの存在しないアドレス領域には物理メモリを割り当てないように SOLID が MMU を設定するので、リード・ライトや命令フェッチ動作のいずれでもプロセッサ例外が発生し、上記同様にバグを自動検出します。

配列オーバーラン

配列として確保したメモリ領域に、配列サイズを超えてアクセスしようとした場合、SOLID が自動的にバグを検出します。

簡単な配列オーバーランの例

```
char buf[10];
for (i=0; i<=10; i++) {
 buf[i]=i;
}
```

この例では 10 個の配列に対してこの for ループを実行すると i が 0～10 まで、11 個のデータを書き込むというバグが含まれています。このプログラムをデバッガを接続した状態で実行すると、11 個目のデータを書き込もうとする直前に、SOLID がバグを検出して"Out of bound access"とサニタイザウインドウにアドレスバグの要因を表示しプログラムをブレークします（図 1-3①）。このとき、アクセスしようとしていた近辺のメモリ内容と、当該命令を各々メモリウインドウ（図 1-3②）と、ソースウインドウ（図 1-3③）でバグの箇所を分かりやすく表示します。

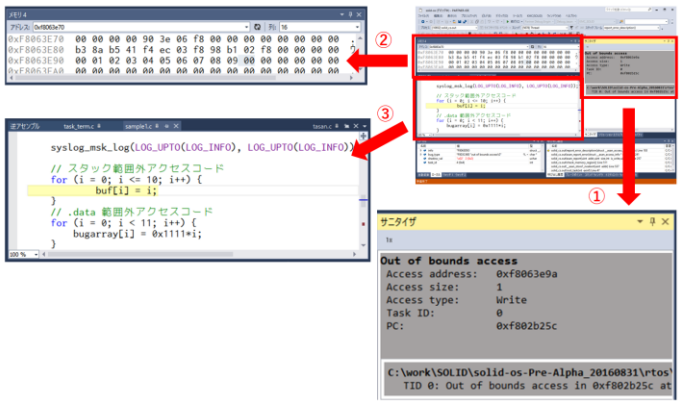


図 1-3.配列オーバーランを自動検出した際のデバッガ表示

配列オーバーランとしては、上記の他にも memcpy()のようなメモリブロック操作命令に対してもバグがあれば自動検出が可能です。

この例で示した配列オーバーランは、SOLID のアドレスサニタイザという専用の実行時バグ検出の仕組みを用いているので、ユーザーが意識的にブレークポイントを設定したり、検出対象の変数を指定する必要は全くありません。SOLID のビルド構成を「アドレスサニタイザモード」に指定するだけの簡単な操作でアドレスサニタイザが使用でき、バグを探し出す時間を大幅に短縮可能です。

ただし、アドレスサニタイザモードを使用する場合、バグ検出のためのランタイムをユーザーコード中に埋め込んで動作させるため、実行時間および使用メモリ（プログラムおよびワーク RAM）にオーバーヘッドが発生します。

このアドレスサニタイザの動作の詳しい仕組みについては、第二章で解説します。

アドレスバグは単体テストでつぶしておきたい

バグを他人に指摘されたくない

アドレスバグのように、「うっかり」作り込んでしまいがちな「単純な」不具合は、出来れば他人に指摘される前に自分でつぶしておきたいものです。

しかし残念ながら自分が作り込んでしまったアドレスバグが不具合として見える形になるのは、他のプログラムと組み合わせてテストしたときなのではないでしょうか。

例えば表示している動画が予期しないタイミングで停止してしまったので、動画処理部を解析していたら、その原因が画像とは全く関係のないネットワーク処理部の配列オーバーによる表示処理部のワークメモリ破壊だった、という事もあり得ます。

SOLID ではバグ動作が起きた瞬間に、他のプログラム動作に何の影響が起きていなくても、アドレスバグを自動検出しますので、アドレスバグを現行犯で逮捕できるところが、従来のモグラたたきでバグを追い詰めるデバッグ方法と大きく異なる点です。

自信をもって単体モジュールをリリースできる

アドレスバグのように、「うっかり」作り込んでしまいがちな不具合を、単体レベルで自動的にあぶりだし事前につぶしておくことでソフトウェアの品質は大きく向上します。開発全体のスケジュールの中で、単体モジュールの検証という比較的初期段階でアドレスバ

グがつぶせるのと、結合テストが進んで様々なバージョンのコンフィグレーションのうち特定の組み合わせで時々発生する不具合症状をデバッグするのでは、デバッグコストも大きく異なります。また悪いことに、時期的にも開発終盤でデバッグにかけられる残り時間が少ない、ということもしばしばです。

特に開発規模が大きくなり、多拠点で分散開発をするような場合は、不具合再現をするだけでも大きな労力を要するものです。委託先協力企業など、社外とのやりとりや段取りだけでも多大な時間を取られてしまうのが現実ではないでしょうか。

SOLID の実行時アドレスバグ自動検出機能を利用し、自信をもって単体モジュールをリリースできれば何よりです。

新しい開発スタイル、SOLID でスマートなデバッグ体験を！

SOLID は、既存の組込み系開発環境に先駆けて、これらのバグ自動検出機能を利用できるようにした初の開発プラットフォームです。テスト用入力パターンをわざわざ作成することなく、簡単な手順でバグ自動検出機能が使えるため、誰もがスマートにデバッグ作業を進める事ができます。

実行時アドレスバグ自動検出機能を使って開発効率をアップし、皆さんのソフトウェア開発パワーをよりクリエイティブな作業に集中していただきたいと思います。

【第二章】アドレスバグ自動検出の仕組み

第二章では、アドレスバグ自動検出の仕組みについてご紹介します。

SOLID では、

Arm® Cortex®-A プロセッサのアクセス例外検出機能
Clang コンパイラのアドレスサニタイザ機能

の 2 つを利用し、更に「SOLID ツールチェーン」、「MMU 設定」と「デバッガ」をシームレスに連携させることで、実行時アドレスバグ自動検出機能を実現しています。本稿では、この 2 つの仕組みについて、少し詳しく解説をしていきます。アドレスバグ検出の仕組みを知ることで、ユーザーの皆様には SOLID を使ったデバッグに対して、「こういうバグはツールに任せれば自動検出できる」という安心や信頼を実感していただけるのではないかと思います。

SOLID には 2 種類の実行時アドレスバグ自動検出機能がある

Cortex-A プロセッサのアクセス例外

Cortex-A プロセッサでは MMU の設定により、ページサイズ（4KB）単位でメモリのアクセス権が設定できます。

「アクセス不可」、「ライト不可」、「実行不可」といった属性を設定した領域に対して、禁止された動作が実行された場合は、プロセッサがアクセス例外を起こします。SOLID デバッガを接続中にこの例外が発生した場合は、デバッガが例外を認識し、アドレスバグが発生したことを GUI 上に表示します。

Clang コンパイラを利用したアドレスサニタイザ

SOLID のもうひとつのアドレスバグ自動検出機能がアドレスサニタイザです。アドレスサニタイザは Google 社の研究チームが LLVM コンパイラ基盤を利用して開発し、Apple 社の統合開発環境 XCode に採用されて普及した実行時バグ検出機能です。コンパイラによる自動コード挿入（instrumentation）で不正なメモリアクセスを検出します。

アクセス例外検出の仕組み

MMU のメモリタイプ属性管理機能を活用する

Cortex-A は MMU（メモリ管理ユニット）と TLB（トランスレーション・ルックアサイド・バッファ）を用いて、仮想アドレスを物理アドレスに変換します。

Cortex-A の MMU では、4KB、64KB、1MB、16MB をサポートするページテーブル エントリを有し、各エントリには、仮想アドレス、ページサイズ、物理アドレスとメモリタイプ属性が格納されます。

このメモリタイプ属性には、次のようなものがあります。

- アクセス不可
- リード可／ライト可
- リード可／ライト不可

これに加えて、実行禁止属性も設定できますので、不当メモリアクセスを検出するには MMU を活用することが有効です。

しかし、Cortex-A プロセッサを使用する場合、MMU の設定が比較的複雑であることに加え、設定を間違えてしまうとそもそもプログラムのロードが出来なかったり、すぐに例外が発生してプログラムが起動しないなど、プログラムを動かすまでに苦労が多いのが実態です。

MMU の活用が有効と分かっていても、難しくて使えない、面倒で使わない、そんな障壁を解消するため、SOLID では誰でも簡単に MMU を使えるようメモリタイプ属性の自動設定機能を設けています。

SOLID による MMU の自動設定の流れ

まずビルド時、ロードモジュールを生成する際にはリンカスクリプトで指示された内容に従い、以下の様なメモリセクション属性や、論理<->物理変換などのテーブルが生成されます。

.text（コード領域）
.rodata（ライト不可のデータ、定数など）
.data（リード／ライト可、実行不可）
.bss（リード／ライト可、実行不可）

SOLID で対象ハードウェア（SoC やボードなど）を選択した時点で自動的にデフォルトのリンカスクリプトが選択されますので、ユーザーは特にリンカスクリプトの内容を意識する必要はありません。

次にプログラムに実行時は、ビルド時に作られた前述のテーブル情報をもとに SOLID が MMU の TLB やページテーブルを設定するためのランタイムを生成し、そのランタイムを初期化処理の一部として実行します。このためユーザーは、自身のプログラムの中で煩雑な MMU の設定が不要になります。

また、論理<->物理アドレスの対応や、メモリセクション属性は、SOLID のメモリマップデザイナで、分かりやすく参照、簡単に変更ができます（図 2-1）。メモリマップデザイナ上で行った変更は、MMU の設定内容にも反映されます。

なおプログラムのリリース時には、MMU 設定内容は SOLID コアサービス（ターゲットの資源管理などを行うソフトウェア）の一部として提供されます。

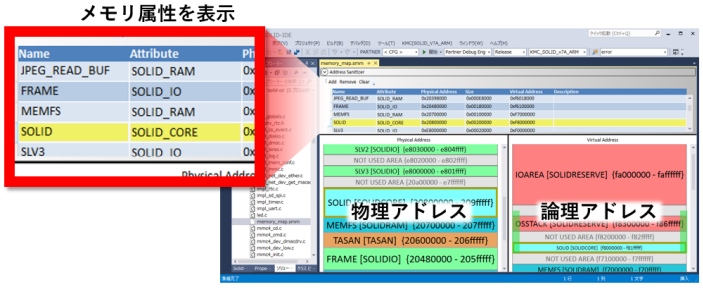


図 2-1. SOLID メモリマップデザイナで、メモリ属性や論理 <－> 物理アドレス対応を参照・変更

このようにセクションの種別に合わせたメモリセクション属性が設定されるため、プログラム実行中に不当なアクセス（ライト不可領域へのライトアクセスなど）があればただちにプロセッサ例外が発生します。プロセッサ例外が発生した場合、SOLID デバッガでは例外の種類を判定し、それをデバッガ画面上に表示します（第一章　図 1-3 でご紹介）。

隙間を空けて論理アドレスにメモリを配置する

SOLID には論理アドレス空間に自動的にセクションデータを配置する機能があります。その際、セクションとセクションの間に隙間を空け、その隙間には物理アドレスを割り当てないことで、プログラムの暴走（図 2-2）やスタックメモリの突き抜け（図 2-3）などを、プロセッサのアクセス例外として検出できるようにしています。

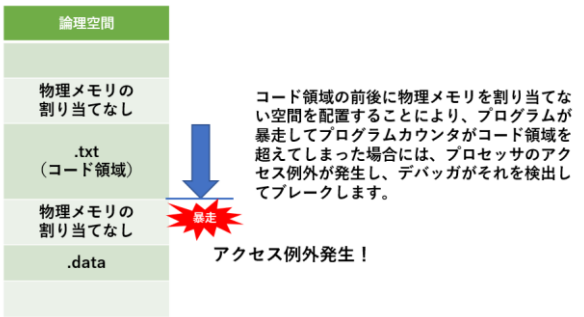


図 2-2. プログラム暴走時は、コード領域境界でバグを自動検出

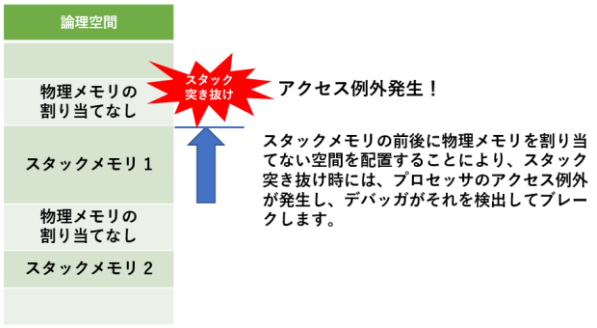


図 2-3. スタック突き抜け時は、スタックメモリ領域境界でバグを検出

Cortex-A　MMU のメモリエントリサイズは、4KB、64KB、1MB、16MB という単位で管理されます。もしプログラムで指定したスタックメモリがこの単位と異なる場合は、SOLID はスタックの上限（スタックを積む方向）にアドレス境界を合わせて配置します（図 2-4）。というのも、スタックメモリに関わるバグの殆どが、スタックの積み過ぎ（想定を超えた多重割り込みなど）が原因だからです。

図 2-4 スタック上限をメモリエントリ境界に合わせて配置する

なお、論理空間上は隙間をあけてメモリを配置しますが、物理空間においては極力隙間を空けずにメモリを配置することでメモリを効率的に利用します。また SOLID がターゲットとしている RTOS ベースのシステムにおいては 1 つの論理空間に全てのメモリを配置することで、メモリスワッピングなどによるリアルタイム性への影響を受けないようにしています。

このアクセス例外によるバグ検出機能においては、実行プログラムサイズおよび実行時間に対するオーバーヘッドは発生しません。

アドレスサニタイザの仕組み

何を監視しているのか?

アドレスサニタイザの機能をシンプルに表現すると次の様になります。

メモリアクセス命令を実行する都度、アクセス対象が妥当かどうかチェックする機構。

図 2-5 の例でその基本的な仕組みを説明します。

プログラムをアドレスサニタイザモード(SOLID のコンフグレーションでは、Debug_Tasan モード)でビルドすると、メモリへのライト命令（ここでは buf[i]=i;）の直前にメモリチェック機能をもつランタイム（以下チェックルーチンと呼びます）を呼び出すコードが自動的に挿入されます。プログラム実行時には、メモリにライト動作が実行される直前に、毎回ライト対象のメモリが妥当かどうかをチェックするプログラムが呼び出され、妥当性が確認されない場合には、プログラムの実行を中断し、デバッガに対して不当アクセスが発生したことを通知します。

（このプログラムは、第一章の解説でも登場した配列オーバーランの例です）

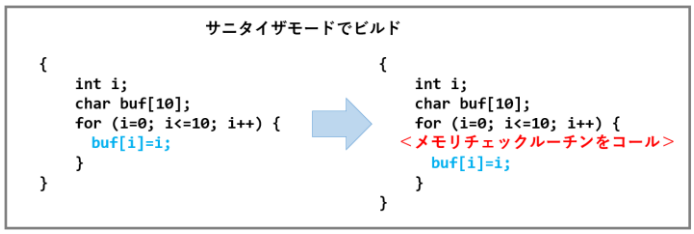


図 2-5 メモリチェックルーチンを呼び出すコードが自動的に挿入される

アドレスバグ発生の可能性がある箇所を、コンパイラが厳選してチェックルーチンをコール

ユーザーはチェックルーチンをコールするコード挿入箇所をわざわざ指定する必要はありません。

というのは、SOLID ツールチェーンに採用している Clang コンパイラにはアドレスサニタイザモードにおいて、コンパイルの際にメモリアクセス命令があるとその直前に予め指定されたチェックルーチンをコールするコード挿入する、という機能を持っており、SOLID がその機能を利用しているためです。

このアドレスサニタイザ機能を使う場合、チェックルーチンの実行時間およびコードサイズのオーバーヘッドを最小限に抑えるため、プログラム中の全てのメモリアクセス命令に対してチェックルーチンを実行するわけではありません。**Clang コンパイラがプログラム**

を解析し、アクセス対象のメモリが論理的に問題ないアドレスにあると判定した場合は、チェックルーチンのコールは行いません。

上記図 2-5.の例のように、変数 i により buf[i]のアドレスが可変となるような、アクセス対象のメモリアドレスが確定しない場合にのみ、チェックルーチンをコールします。逆に、i についてはメモリアクセスが発生する可能性があるものの、問題のないアドレスに配置されていることをコンパイラが知っているので、i のアクセス直前にはチェックルーチンをコールするコードを挿入せずにプログラムを実行します。

チェックルーチンがやっていること

Clang コンパイラがアドレスサニタイザモードで実行するチェックルーチンは、SOLID が専用に用意したデバッグ専用のランタイムであり、次のような動作をしています。

まず、アドレスサニタイザモードでビルドすると、論理空間上に変数を配置する際にその前後に「ガード領域」を挿入します（図 2-6）。ガード領域には物理アドレスが割り当てられていても構いませんが、プログラムの動作においてはアクセス無効と定義されます。

図 2-6. 変数の間にガード領域を挿入

次にプログラムを実行すると、アドレスサニタイザの監視対象関数の先頭で、アドレスサニタイザ専用のランタイムが論理空間のアドレス 1 バイトに対して 1 ビットの有効無効ビットを持った、アドレス判定テーブルを設定します。対応するアドレスが無効であれば”0”、有効であれば”1”という値を持ったテーブルです（図 2-7）。

プログラムの進行に従いチェックルーチンが実行されると、アクセスしようとしているアドレスの有効・無効をアドレス判定テーブルを参照して判定します。判定結果が有効であれば、チェックルーチンは何もせずに、続くメモリアクセス命令を実行します。一方、判定結果が無効の場合は、アクセス違反を検出し、デバッガに通知するためブレークを発生させます。

図 2-7 において、buf[i]の i=10 になったところで判定テーブルの内容が 0 となるため、チェックルーチンがこれを不正アクセスと判定します。

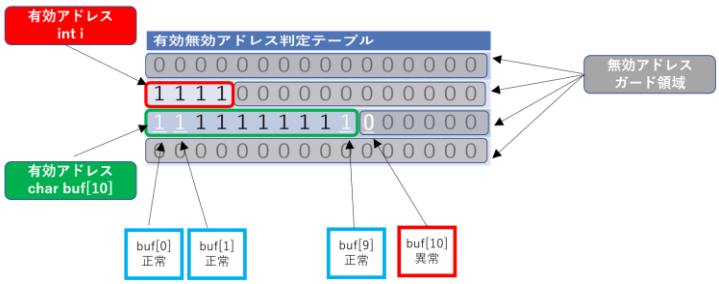


図 2-7　有効無効アドレス判定テーブルを利用して、
不正アクセスを判定する

少々細かい説明となりますが、この有効無効アドレス判定テーブルは、ターゲットメモリの中の “SOLID-RAM”という領域内に確保されます。

アドレスサニタイザモードでのバグ検出機能の仕組み（まとめ）

アドレスサニタイザによる実行時アドレスバグ自動検出の仕組みをまとめると、以下のようになります。

1. コンパイラがプログラムを解析して不正アクセスが発生する可能性のある個所にチェックルーチンを呼び出すコードを埋め込む
2. コンパイラがガード領域を挟んで変数を配置
3. 実行時、関数の先頭で有効・無効アドレス判定テーブルを設定
4. メモリアクセス命令が実行されると、SOLID が用意したランタイム上で動作するチェックルーチンがメモリアクセスの都度判定テーブルでメモリアドレスの有効・無効をチェック
5. アクセス対象が有効なメモリであれば、そのままプログラム実行を継続
6. 無効アドレスへのアクセスを検出したらデバッガが IDE 経由でユーザーに通知

このように、コンパイラ、デバッガ、IDE と SOLID 専用のランタイムが密接に連携してはじめて実現できるアドレスバグ自動検出機能を、SOLID ではアドレスサニタイザモードでビルド＆実行するだけで誰でも簡単に利用できるようになっています。

このとき、変数の配置や、チェックルーチンの呼出しコード挿入も全て SOLID が自動的に行うため、ユーザーがアドレスバグの発生しそうな箇所を意識する必要はありません。

なお、アドレスサニタイザモードでは、変数間のガード領域、有効無効アドレステーブル、チェックルーチンやチェックルーチンを呼び出

すコードの挿入によりメモリ容量のオーバーヘッドがあります。また、チェックルーチン実行による実行速度低下が生じる点にも留意が必要です。

第三章では、実行時アドレスバグ自動検出機能を有効に利用していただくためのヒントなどをご紹介していきます。

【第三章】
アドレスバグ自動検出の有効な使い方

第一章、第二章では、実行時アドレスバグ検出機能がどのようなバグをみつけられるのか、どのような仕組みでバグを検出しているのかについて、ご紹介してきました。本章では、アドレスバグ自動検出機能を利用する際の留意事項や、有効な使い方、活用例などを中心にご紹介していきます。

アドレスバグ自動検出機能は万能ではない

検出しないケースもある

SOLID では、実行時アドレスバグ自動検出の仕組み上、次のようなケースでは意図しないメモリアクセスがあっても、それをバグとして検出しません。

スタック領域の目的外アクセス

スタックオーバーフローの自動検出は、予め設定されたスタックメモリを突き抜けて、メモリが存在しない領域にスタックを積もうとして例外となりバグを検出する仕組みですが、スタック操作以外の動作によりプログラムがスタック領域をアクセスした場合、そこが物理メモリが割り付けられている論理アドレスであれば、この動作をバグとして検出しません。スタック領域のアクセスが不当なのか正当なのかはプログラムの仕様そのものに依存する問題であり、意図的に操作する事もあるので、メモリアクセス命令はそのまま実行されます（図 3-1）。

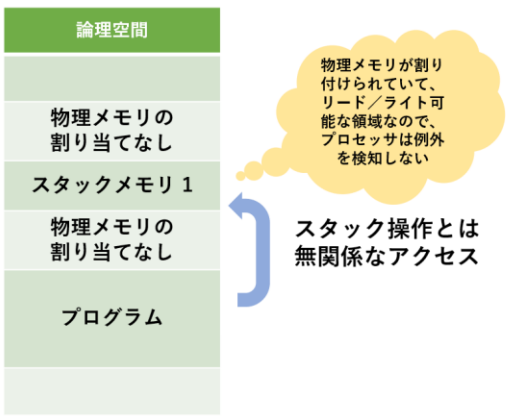


図 3-1. スタック領域の目的外アクセス

不連続な配列オーバーラン

配列アクセスにおいて配列の範囲をはみ出した場合には、アドレスサニタイザにより無効アドレスであると判定されてバグ検出ができます（検出の仕組みは第二章を参照してください）。

しかし、図 3-2 のように配列の要素を不連続に（ガード領域を飛び越えて）アクセスした場合、たまたまそのアドレスに他の変数が割り当てられていれば、配列としては正しくありませんが、アクセス対象が有効アドレスなのでアドレスサニタイザ機能が有効であってもこれをバグとして検出することが出来ません

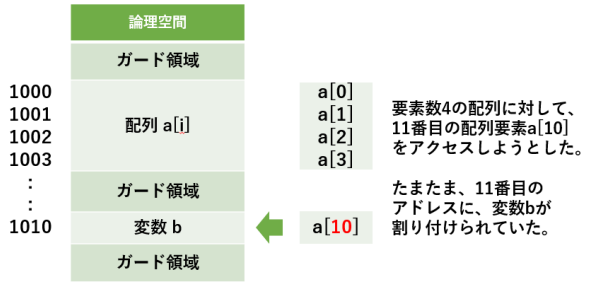


図 3-2. 不連続な配列オーバーラン

もっとも、このようなプログラムを書いてしまった場合は、SOLID の静的解析機能が論理的なバグとして指摘する可能性が高いので、実行前に不具合が判明することがほとんどでしょう。

仕様上ライト不可な変数は const 宣言しておくこと

仕様上ライト対象ではない変数に対して、意図しないプログラムが誤ってライト動作をした場合でも、アドレスサニタイザはこれをバグとして検出することはありません。なぜなら、アドレスサニタイザでは対象アドレスの有効・無効のみを判断しておりリード・ライトといったアクセス種別の判定はしていないからです。

変数を確実にライト不可としたい場合は、const 宣言により定数に指定してください。const 宣言した変数はセクションのメモリ属性がライト不可な .rodata に領域に割り当てられるため、その変数へのライト時にアクセス例外が発生し、確実にバグとして検出可能です。

オーバーヘッドにも注意

アドレスサニタイザモードでデバッグをする場合、以下の理由により図 3-3 に示すようなメモリ消費量のオーバーヘッドが発生します。

- アドレス判定のためのチェックルーチンを呼ぶためのコード（図 3-3 ①）
- チェックルーチン（図 3-3 ②）
- 変数の前後のガード領域（図 3-3 ③）
- アドレス有効無効テーブル（図 3-3 ④）

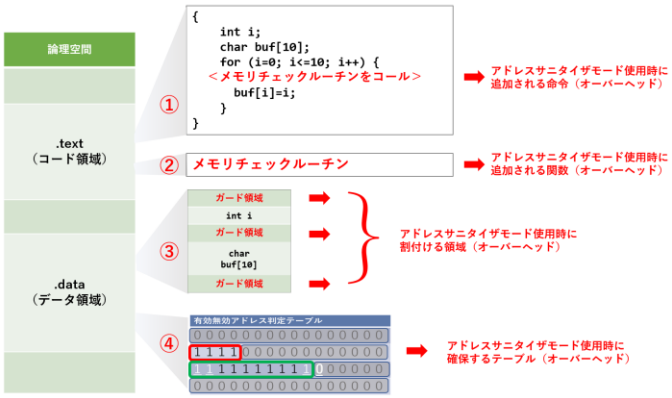


図 3-3 アドレスサニタイザモード使用時の
メモリ消費量オーバーヘッド

プログラムの内容によりオーバーヘッドは大きく変わりますが、アドレス有効無効テーブルは 1 バイトあたり 1 ビット必要なので、変数領域の 8 分の 1 がオーバーヘッドになります。また内部変数に関しては、ガード領域がスタック使用量増加になるので、スタックオーバーフローが起きないよう、スタックサイズを 2 倍程度確保する必要があります。その他、変数間のガード領域やチェックルーチンを呼ぶコードのオーバーヘッドもあるため、全体として、アドレスサニタイザを使わない時に比べて 1～3 割程度多くのメモリを消費します。

また、変数アクセスの都度チェックルーチンが実行されるため、一般的なプログラムにおいてアドレスサニタイザモードで実行した場合は、通常時に比べて実行時間が平均すると 2 倍程度遅くなります。

なお、MMU を利用したメモリ属性の違反検出（スタック突き抜けや、const 宣言変数へのライト動作）においては、メモリ容量や実行時間に対するオーバーヘッドは発生しません。

実行時アドレスバグ自動検出機能の有効な使い方

検出対象を指定することで、オーバーヘッドの影響を少なく

上記のように、アドレスサニタイザモードでの動作時にはメモリの消費量および実行時間に対するオーバーヘッドがあるため、制御対象によっては正常に動作しない場合もあると思います。しかし、アドレスサニタイザモードでは、「特定のプロジェクトだけ」「特定の関数だけ」を検出対象に指定できるという便利な使い方が可能です。

そのため、プロジェクト単位でアドレスサニタイザを利用したり、個別の関数単位でアドレスサニタイザモードを用いて予め検証し、結合時テストではアドレスサニタイザモードを全て外す、といった使い方でオーバーヘッドの影響を少なくすることが可能です。

一般的なテスト検証においては、バグがどこに影響を及ぼすかがわからないため、ソフトウェア全体を結合した状態で影響範囲の検証を行う必要があります。しかしアドレスサニタイザでは影響を及ぼされた側ではなく、影響を及ぼした側の不正アクセスを「現行犯逮捕」できる仕組みなので、部分的に分割してアドレスサニタイザを適用したデバッグでも、不正アクセスの検出に十分有効です。

チーム開発時のリビジョンアップ作業も安心

ソフトウェアの規模が大きくなりチーム体制で開発する場合、担当モジュールを更新してリリースする前にアドレスサニタイザモードを有効にして実行テストをしておけば、他のモジュールへ影響を与えないことが確認できるため、安心してモジュールの更新＆リリースが出来ます。

アドレスサニタイザ機能の SOLID 流アレンジ

memset()、memcpy()関数では効率よくメモリをチェックする

アドレスサニタイザモードにおいては、メモリアクセス命令を実行する度に対象のメモリが有効か無効かを判定するチェックルーチンを実行しています。そのため、memset()や memcpy()といったメモリブロック操作関数では操作対象の全アドレスに対してメモリアクセスの都度 有効無効チェックが実行され、実行時間のオーバーヘッドが大きくなってしまうという懸念があります（図 3-4　左）。

そこで SOLID では独自の機能で対処しています。まず、SOLID で用意しているライブラリの memset()および memcpy()で行われるメモリアクセスは、アドレスサニタイザの対象から除外されています。

そのうえで、関数の先頭で一度だけ操作対象アドレスの全範囲が有効か無効かをチェックし、一部でも無効アドレスであれば、アドレスバグとして検出を行うよう、SOLID 独自の API を用意してオーバーヘッドを少なくしています。逆に全域が有効アドレスであればメモリブロック操作命令を実行します。（図 3-4 右）。

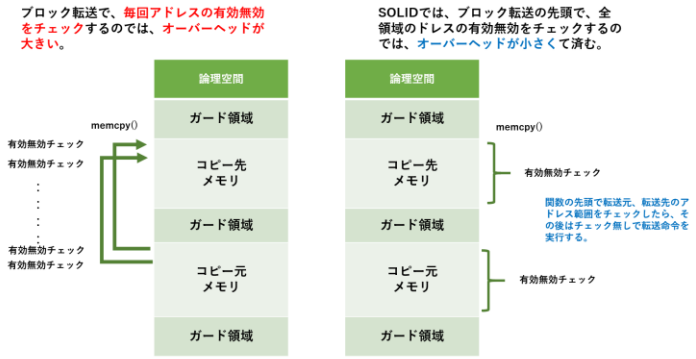


図 3-4 メモリブロック操作命令では、
SOLID が独自にアドレスチェック回数を少なくする

ここで使用した「特定の命令や関数をアドレスサニタイザの対象から除外する機能」は、Clang コンパイラが持っている機能です。この機能と、SOLID 独自の「API を活用してアドレスサニタイザのアドレスチェック回数を削減する仕組み」を活用することで、メモリブロック操作におけるアドレスチェックルーチンの実行回数（実行時間のオーバーヘッド）を大幅に削減することができます。

malloc、free 関数によるメモリの動的割り当ても監視

SOLID 独自の機能として、malloc、free 関数によるメモリの動的割り当ても監視しています。

アドレスサニタイザモードでは、malloc 関数が実行された場合に、当該アドレス領域に対してアドレス有効・無効テーブルを「有効」にセットし、free 関数が実行されたら、その領域は「無効」にします。また、malloc、free 関数の先頭では、アドレス有効・無効テーブルのチェックをするルーチンを追加します。

プログラムを実行中、free 関数実行の際に、既にその領域が「無効」になっていた場合は、「二重解放」に当たるため、SOLID はこれを不正動作として検出し、プログラムをブレークしてデバッガを通してユーザーに通知します。

同様に、free 関数実施後に当該アドレス領域をアクセスした場合も、不正動作として検出します。

ブレークした箇所を分かりやすく表示

これまでご紹介してきたように、アドレスサニタイザモードを使ったデバッグではユーザーが作成したプログラムに対して、不正検出やデバッグのためのコード（ランタイム）が多数追加されて走っています。

つまり、アドレスバグを自動検出してプログラムがブレークした瞬間の PC(プログラムカウンタ)は、ユーザーのコードとは別の不正検出やデバッグのためのコードの中にあることになります。

この PC はデバッグ対象であるユーザー自身のコードの中ではないため、デバッガの画面ではブレークした PC の箇所ではなく、不正の元となったユーザープログラムのソースコードの箇所を分かりやすく表示する工夫をしています。またトレース履歴にはブレークするまでに走ったコードのトレース情報が不正検出用のコードを含めて記録されていますが、不正が発生したユーザーコードの箇所が判るように表示をする、といった配慮もしています。

利用者の体験から

最後になりますが、SOLID の実行時アドレスバグ自動検出機能を使って便利にデバッグができた体験談を紹介いたします。

ソースコード無し、バイナリで入手したライブラリの実装デバッグ

ある SoC を搭載したボードに、外部から提供された画像処理ライブラリのポーティング作業をすることになりました。画像処理ライブラリはバイナリだけの提供で、ソースコード無しという条件でした。

SOLID 環境でライブラリのポーティング後にデバッガを接続してプログラムを実行させたところ、プログラムがブレークし、SOLID IDE がスタックオーバーフローを表示しました（図 3-5 オーバーフローしたタスクのスタック使用量が赤で表示されるため、すぐに気付くことができました)

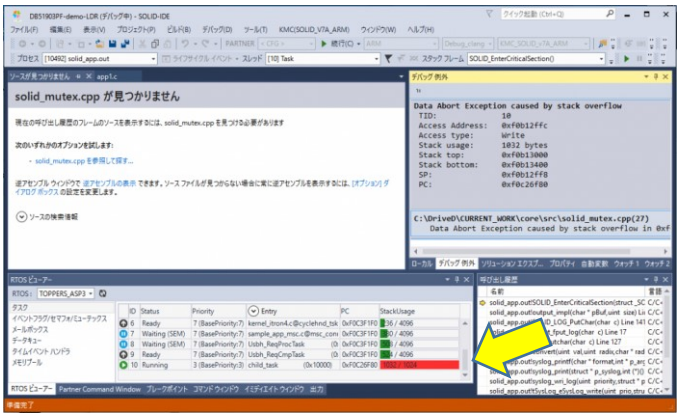


図 3-5 スタックオーバーフロー発生時のデバッガ画面

RTOS ビューアでスタックオーバーフローを起こしたタスクを確認し、そのタスクのスタックサイズを 2 倍に設定して再試行してみたところ（ソースコードが無くてもスタックサイズは変更可能です）今度はエラーの発生はなく、無事にポーティング作業が終了しました。

このように、デバッグ担当者が意図的にブレークポイントを張るという操作は行わず、ソースコードも無い状態でしたが、即座に不具合箇所が見つかり対策する事が出来ました。

なおこの例では、テスト用の機能を有効にしてビルドした場合に、通常時よりも多くのスタックメモリを消費する機能が動作するのが直接的な原因でした。すなわちコードに本質的な問題があったわけでなく、いわば設定ミスのようなものでした。

従来の RTOS での開発においては、このようなトラブルのときには、スタックを破壊してから暴走するなり不正な動作になってしまうので、原因を一瞬で突き止める、ということはなかなかできませんでした。過去の経験で「スタックサイズが原因かも？」と思うことができれば早期解決は可能ですが、そうでない場合には、どこまで正しく動作したか、ということをブレークポイント設定したり、またログを仕込んだりするところからデバッグを始めなければなりません。設定ミスであっても通常のデバッグ並みに多くの時間を消費することもあるのです。しかし SOLID のスタックオーバーフロー自動検出機能では、このようにスタックが足りなくなった状況を即座にユーザーに通知するので、無駄な時間を消費することがありません。

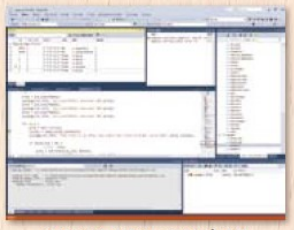
最後に

従来のデバッグ手法では、正当なメモリの利用者が実害を受けてから不具合現象がわかり、そこから真犯人を探す旅が始まるわけですが、SOLID の実行時アドレスバグ自動検出機能を活用することで、不当アクセスが起きた瞬間に現行犯で犯人を逮捕できるのです。本書を通して、皆様にもその違いの大きさがお分かりいただけたことと思います。

（終）

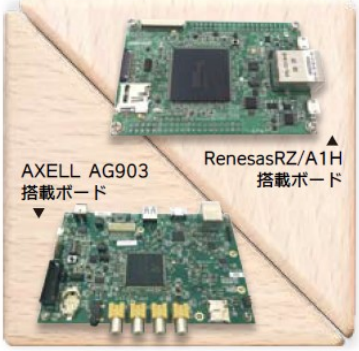
¥50,000-で、SOLIDを体験してみよう!

SOLID スターターキット



SOLID-IDE, コンパイラ
RTOS 込みの付属ボード用BSP

PARTNER-Jet2 Model10



IDE,コンパイラ,JTAG,ボードと、PC以外の必要な物全てが含まれます。
イーサネットドライバなどのBSPも付属して、購入してスグにSOLIDの開発環境を体験できます!!

CPUの異なる
二種類のボードを
用意しました。

Renesas RZ/A1H

- 名刺サイズ、バスパワー動作
- ARM **Cortex-A9** 400MHz
- 10Mbyte内蔵RAM
- アルファプロジェクト社製

AXELL AG903

- 豊富な映像処理機能
- ARM **Cortex-A5** 400MHz
- 64Mbyte内蔵RAM
- アクセル社製

スターターキットでは、SOLIDが提供する開発環境の様々な機能の全てを体験いただけます。

- Visual Studio®ベースのSOLID-IDE
- コンパイラによるソースコードの静的解析機能
- アドレスサニタイザによるメモリバグ自動検出
- MMUを効果的に使ったプロテクションとローダー機能

SOLID Starter Kit for RZ/A1H
SOLID Starter Kit for AG903

どちらも ¥50,000（税抜き）

本キット同梱の開発環境は、購入後6ヶ月間利用可能です。ボードの利用期間に制限はありません。開発環境の利用期間については、追加オプション(有料)で期間制限解除ができます。キット同梱の開発環境で生成されるソフトウェアは、付属ボード専用です

この記事はこちらからもお読みいただけます。

京都マイクロコンピュータ株式会社
本　　社:〒610-1104 京都市西京区大枝中山町2-44　Tel.075-335-1050
東京オフィス:〒105-0004 東京都港区新橋2-14-4 Rビル5F　Tel.03-5157-4530

SOLID製品情報はこちら
https://solid.kmckk.com/SOLID/

2018.6(第 1 版)